The Paradigm Change from Thin to Rich Clients in Modern Web Applications

An analysis of how the MVC pattern is being shifted to the client side

for the

Master of Science

in Interaction Design

at the Baden–Wuerttemberg Cooperative State University Stuttgart

by

Tim von Oldenburg

September 2012

Time of Project	10 weeks
Student ID, Course	0834311, TIT09AIA
Company	IBM Deutschland MBS GmbH, Herrenberg
Supervisor in the Company	Lars Holmberg
Reviewer	Paul Hubert Vossen

Abstract

Research Question: *How can an IDE support the understanding and exploration of scope & context in a programming language to deepen understanding and help prevent errors?*

- IDE giving semantic support in code creation
- Understanding asynchronous control flow in IDEs // nope!!!
- Understanding and exploring scope and context
- Goals: understanding, less errors/mistakes

Acronyms

IDE Integrated Development Environment

PARC Palo Alto Research Centre

UI User Interface

WYSIWYG What You See Is What You Get

Contents

1 Introduction

Lorem Ipsum.

2 Research, Framework

This chapter will introduce the research done prior to the design. It will explain the motivation behind working on software development environments, give a short history of Integrated Development Environments (IDEs) and list typical User Interface (UI) design patterns in IDEs. It will close by presenting a survey and a series of interviews done in order to understand the problem space.

2.1 History and role of IDEs

Software development environments have been predecessed by general text editors, starting with several projects at the Xerox Palo Alto Research Centre (PARC). Douglas Engelbart created the text editor for the NLS system (oNLine System) which allowed What You See Is What You Get (WYSIWYG) style editing (Moggridge 2007, pp.). In the *Gypsy* text editor, Larry Tesler first integrated modeless moving of text, which is known as *Copy&Paste* (Moggridge 2007, pp.). Text editors with those functionalities are now the core of any software development environment.

Later, while working with Alan Kay, Tesler created the first class browser for the Smalltalk programming language. Class Browsers are used to look at source code not as textual files, but as logical entities of a programming language (for example, classes and methods). The Smalltalk class browser was therefore the first software specifically written for creating software, and a predecessor to any modern IDE.

IDEs integrate text editors (due to their specific purpose also referred to as *code editors*) with other software development tools. Typically, those tools may include compiler, build system, syntax highlighting, autocompletion, debugger, and symbol browser.

Nowadays, IDEs make use of many more UI patterns and adapt them to a specific purpose. Taking the Eclipse IDE as an example, one can see that the class browser is built using a Tree View (as

often seen in file browsers), and the text editor uses bold, italic and coloured text automatically to distinguish different entities of the programming language (so-called *syntax highlighting*).

TODO: screenshot of eclipse w/ class browser + syntax highlighting

The IDE landscape is today more differentiated than ever, ranging from minimal, purpose-specific environments like Processing to huge, general-purpose, commercial environments like Visual Studio. Those different IDEs serve the needs of different developers and development situations. But still, it seems like there are many niches that are yet to be filled with new IDEs. Especially the area of web development (frontend development) is seeing many newcomers, for example Github's Atom Editor, Adobe's Brackets and Eclipse Orion, all based on Node.js and other web technologies.

2.2 UI and Interaction Patterns in IDEs

As previously mentioned, most UI patterns found in IDEs are general, well-known patterns adapted to a specific purpose. This section will give an overview on relevant interaction patterns in IDEs and their graphical implementation.

2.2.1 UI Patterns

Code Editor Central to every IDE, a code editor is a specialized text editor, used for reading and writing program code. It usually features a *gutter* (see below) and Syntax Highlighting. In opposition to the text editor of a word processor, code editors usually display a monospaced font, which allows to see the code editor as a grid of rows and columns. With evenly-spaced columns, due to the monospaced font, code formatting is made consistent; line indentation is an important concept in many programming languages, either as a core syntactical concept or for the sake of readability.

Gutter The gutter is part of the code editor and describes the narrow space next to the actual code (usually to the left). Gutters are mainly used to display line numbers (important for navigation and debugging), but some provide more advanced features, for example setting breakpoints[1], indicating errors in the code through symbols or showing version control information.

[1] A feature of the debugger; when set, the program stops at the specified line to allow step-by-step investigation.

- (Inline) popup

Panel (sidebar) A panel is rectangular UI element used to group interface element of similar functionality together. Often, panels **TODO: moar**

Status bar The status bar is known from many programs, for example web browsers and word processors. It is a small bar (about one text line of height) at the bottom of the program window, usually spanning the whole window width. It is mainly used to display status information and quickly switch between different modes.

2.2.2 Interaction/behavioral patterns:

- Navigation
- Editing
- Reading/understanding
- Exploration
- Mouse and keyboard (shortcuts) as input
- Modes (vim, larry tesler against modes, diff. configurations in eclipse, on-the-fly hide/show in sublime text/atom etc)

2.3 Relevant programming concepts

The following section presents concepts of programming and programming languages that are important to the topic of this thesis. Whereas most of the concepts apply to a wide range of programming languages, *JavaScript* was chosen as an exemplary language both to explain the concepts as well as the target language of prototyping as described in the next chapter. The reasons for this choice are the author's familiarity with the language, as well as the fact that is one of the most ubiquituous languages used due to its role in the world wide web and its implementation in web browsers, respectively.

2.3.1 Run time

In the lifecycle of a program, run time is the phase in which a program is executed

moar

2.3.2 Syntax & Semantics

2.3.3 Scope & Context

(Simpson 2014)

> Just as a block or function is nested inside another block or function, scopes are nested inside other scopes. So, if a variable cannot be found in the immediate scope, Engine consults the next outer containing scope, continuning until found or until the outermost (aka, global) scope has been reached.

- Function scope
- Global scope (most outer scope)
- Start from local scope (where the statement is defined), and work your way outside nested scope
- Functions define a new scope; blocks do not (in JavaScript)
- Scope: a set of rules to look up variables and their values.

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up. This look-up may be for the purposes of assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-side) reference.

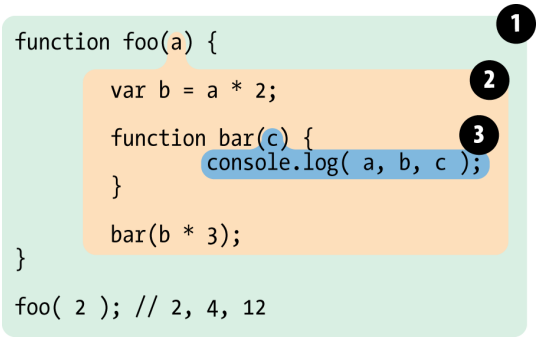

Figure 2.1: Scope as bubbles (Simpson 2014)

2.4 Survey & Interviews

To form a general understanding of how IDEs and some of their specific features are used, an online survey targeted towards professional developers was created. The survey ran in April 2014 over the course of two weeks and yielded answers from 45 participants.

Besides general questions, e.g. which programming languages and IDEs the participants used, it collected information about the usage of the following IDE functionalities:

- Navigation of code
- Debugging
- Usage of API and language documentation
- Autocompletion
- Project structure and scaffolding
- Asynchronicity
- Syntax Highlighting

For each of the areas it was asked if and how the participants were using them and—if appropriate—how their IDE was supporting them. They survey instrumented multiple-choice questions with an additional „Other“ field for custom answers, as well as open-ended questions with a free-form text field.

3 Design & Process

3.1 Process

- Ideation & Concept
- Prototyping: esprima, atom, brackets, devtools?
- User Testing, Probe (1 week)
- Interview, evaluation in quantity and quality

3.2 Existing solutions & inspiration, similar solutions (IDEATION)

- Syntax Highlighting (similar)
- Indentation (similar)
- Crockford's context coloring (existing, inspiration)
- Theseus' grey colouring for code that hasn't been called

3.3 IDEATION

To support the ideation phase, the author created a collection of existing UI components within IDEs. Those components were written down on post-it notes.

The components were used a seeds for *seeded brainstorming*: for each of the components, the author tried to imagine solutions that are similar, related to or based on the respective component.

Most of the ideas that came out of the brainstorming session made use of multiple components, for example the *context path* which is described further down: it made use of a status bar as well as the code editor.

Most of the ideas of the brainstorming phase made it into first sketches. The sketching happened with two different approaches, depending on if the code editor was involved or not.

For ideas that involved the code editor, it was important that the author could work with real, functioning code. Therefore, two sample JavaScript applications were created to work with:

- A small web server application, that would parse a markdown-formatted text file and render it into an HTML template. The application would run on Node.js and represents a typical control flow for e.g. a blogging engine (content + template = site).
- A client-side script (runs in a browser) that fetches JSON data and presents them on a website, by the click of a button. This script represents typical client-side UI code, connecting a button event to a function and presenting the result in the UI.

Both applications were written in different styles: the server-side application decouples the different tasks by putting them into different functions (as far as it makes sense), whereas the client-side application nests all function definitions inside each other, resulting in nearly one function definition in each line, and deeper indentation (ergo: higher code complexity).

A good solution for this design problem should address both cases.

Printouts of the two JavaScript files served as a basis for ideation *within source code*.

For concepts that would mainly work with other UI components, such as a sidebar, or such concepts that would introduce new UI components, blank paper was used for sketching.

3.4 CONCEPTS

- *Context Path* - a path view of the context tree, similar to that of a selector path in an HTML editor (screenshot!). The context at the position of the cursor would be shown in a status bar. By hovering over a context level, the corresponding source code would be highlighted in the source editor.
- *Context Graph* - similar to a class browser, the context graph would represent a tree view of the application's context(s). This could be implemented as a sidebar or panel.

- *Context Colouring* - similarly suggested by Crockford, the source code can be coloured in depending on its context (level). Crockfords variation is meant to replace syntax highlighting; one could, as well, complement syntax highlighting by colouring in the background (as e.g. Theseus does). 50 Shades of Grey.
- *Inspect Context* - comparable to DevTools' *Inspect Element* function, the user can right-click into the source code and choose *Inspect Context*, which opens a panel that shows global variables, current local variables as well as the value of `this`.
- *Gutter Context* - any change of context or scope is indicated in the code editor's gutter (similar to JSHint).
- *Quick Inspect* - similar to Brackets' *Quick Edit* feature, the value of `this` could be inspected inline.

3.5 PROTOTYPING

3.6 USER TESTING, (Probe) (1 week)

4 Conclusion, Reflection

Bibliography

Moggridge, B. (2007), *Designing Interactions*, The MIT Press, Cambridge, MA, USA.

Simpson, K. (2014), *You Don't Know JS: Scope & Closures*, O'Reilly Media.